AFRL-AFOSR-VA-TR-2015-0248

Virtual Machine-level Software Transactional Memory: Principles, Techniques, and Implementation

Binoy Ravindran
VIRGINIA POLYTECHNIC INST AND STATE UNIVERSITY

08/13/2015
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTC
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 10-08-2015	**2. REPORT TYPE** Final Report	**3. DATES COVERED** *(From - To)* July 1, 2014 -- June 30, 2015

4. TITLE AND SUBTITLE Virtual Machine-level Software Transactional Memory: Principles, Techniques, and Implementation	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER AFOSR FA9550-14-1-0143
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Ravindran, Binoy	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Virginia Polytechnic Institute and State University, 1880 Pratt Dr, Suite 2006, Blacksburg, VA 24060-3580	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research (AFOSR), 801 N Randolph St., Arlington VA 22203, Program Officer: Dr. Kathleen Kaplan, System and Software, RTA2-9, Phone: 703-696-7312, kathleen.kaplan@us.af.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFOSR
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution A -- Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software transactional memory (STM) has emerged as an easy to program synchronization abstraction for multicore computer architectures. But performance of current STM frameworks are inferior and heavily influenced by infrastructure resource management (e.g., operating system scheduling, garbage collection, memory allocation). The project's first major result is ByteSTM, a virtual machine-level Java STM implementation that is built by extending the Jikes RVM. The project modified Jikes RVM's optimizing compiler to transparently support implicit transactions. Being implemented at the VM-level, which enables direct memory accesses, ByteSTM avoids Java garbage collection overhead by manually managing memory for transactional metadata, and provides pluggable support for implementing different STM algorithms. Three well-known STM algorithms have been integrated into ByteSTM: TL2, NOrec, and RingSTM. The project's experimental studies revealed throughput improvement over other non-VM STMs by 6–70% on micro-benchmarks and by 7–60% on macro-benchmarks. ByteSTM is open-source, publicly available (http://hydravm.org/bytestm/), and is used by the TM community.

15. SUBJECT TERMS

Concurrency, synchronization, transactional memory, multicore

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Binoy Ravindran
U	U	U	UU	1	**19b. TELEPHONE NUMBER** *(Include area code)* 540-231-3777

Reset

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATES COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33615-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report, e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5d. PROJECT NUMBER. Enter all project numbers as they appear in the report, e.g. 1F665702D1257; ILIR.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organization, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.

Virtual Machine-level Software Transactional Memory: Principles, Techniques, and Implementation

Final report for AFOSR Grant FA9550-14-1-0143
PI: Binoy Ravindran
ECE Department, Virginia Tech
302 Whittemore Hall, Blacksburg, VA 24061
Phone: 540-231-3777, Fax: 540-231-3362, E-mail: binoy@vt.edu

Abstract

Software transactional memory (STM) has emerged as an easy to program synchronization abstraction for multicore computer architectures. But performance of current STM frameworks are inferior and heavily influenced by infrastructure resource management.

The project's first major result is ByteSTM, a virtual machine-level Java STM implementation. Being implemented at the VM-level, which enables direct memory accesses, ByteSTM avoids Java garbage collection overhead by manually managing memory for transactional metadata, and provides pluggable support for implementing different STM algorithms. The project's experimental studies revealed throughput improvement over other non-VM STMs by 6-70% on micro-benchmarks and by 7-60% on macro-benchmarks. ByteSTM is publicly release as open-source.

The project's second major result is a set of theoretical results on ensuring the Disjoint-Access Parallelism (DAP) property for STM implementations. Assumed a DAP TM that provides invisible and wait-free read-only transactions, the project proved that relaxing the Real-Time Order (RTO) is necessary. This result motivated the definition of Witnessable Real-Time Order, a weaker variant of RTO that restricts RTO to directly conflicting transactions only. Finally, the project established lower bounds on the time and space complexity of DAP TM implementations.

I. Overview of Project Achievements

This project investigates theoretical and practical aspects of Software Transactional Memory (STM) implementations. STM is a growing programming abstraction for developing concurrent applications. With STM, the complexity of synchronizing multithreaded access to a set of shared data (or objects hereafter) is no longer a burden for the application programmer; they can simply focus on the application business logic and rely on the STM library to guarantee correct concurrent execution. Thanks to STM, the effort of designing, developing, and maintaining a concurrent, multithreaded application is significantly alleviated.

The project's most significant accomplishment towards the practice of STM is the design and implementation of ByteSTM [31], [30], a virtual machine-level Java STM implementation built by extending Jikes RVM. In ByteSTM, a transaction is not restricted to methods and can surround any block of code. Memory bytecode instructions reachable from a transaction are translated such that the resulting native code executes transactionally. Given its JVM integration, ByteSTM can access, allocate, and manage memory directly, thus overcoming restrictions of the current JVM design. ByteSTM has a modular architecture, which allows different STM algorithms to be easily plugged in (three well-known algorithms have been already: TL2 [12], RingSTM [37], and NOrec [11]). The conducted experimental study revealed a 6-70% throughput improvement over other non-VM STMs on micro-benchmarks and a 7-60% improvement on macro-benchmarks.

ByteSTM is open source and is freely available at `http://hydravm.org/bytestm`. It provides usable and high-performance STM support for Java applications, and has been transitioned to the STM community at large with an increasing number of citations (e.g., [15], [36]). It contributes to the broad dissemination of the STM abstraction in the (not only research) community.

The project's most significant accomplishment towards the theory of STM is a set of results regarding the impossibility, possibility, and inherent cost of STM implementations that provide the Disjoint-Access Parallelism (DAP) property [32]. DAP is considered one of the most desirable properties for providing scalable STM algorithms; roughly, two concurrent threads are allowed to conflict on some shared resource (e.g., objects or meta-data) only if the application itself demands that. Given its appealing purpose, this project has investigated what can (and cannot) be done while preserving DAP. First, it was proven that relaxing Real-Time Order (RTO) is necessary for an STM implementation that ensures DAP as well as two properties that are regarded as important for maximizing efficiency in read-dominated workloads; namely, having invisible and wait-free read-only transactions. This result provided the base for introducing Witnessable Real-Time Order (WRTO), a weaker variant of RTO that demands enforcing RTO only between directly conflicting transactions. WRTO makes it possible to design a strictly DAP STM with invisible and wait-free read-only transactions while preserving strong progressiveness for write transactions and an isolation level known in literature as Extended Update Serializability [2] and establishes a lower bound on the time and space complexity of DAP STM implementations that have invisible and wait-free read-only transactions.

The project's findings enrich the common knowledge of DAP STM implementations and provide a better understanding for STM designers about the inherent costs and limitations of adopting DAP.

The rest of this report is organized as follows: in Section II we detail ByteSTM, the major achievement towards the practice of STM; in Section III we provide the description of the major accomplishments towards the theory of STM; and in Section IV we conclude the report by summarizing the project's deliverables (i.e., software and publications).

II. ByteSTM, a Java virtual machine-based STM implementation

Transactional Memory (TM) [21] is an attractive programming model for multicore architectures that promises to help the programmer in the implementation of parallel and concurrent applications. TM has been proposed in hardware (HTM, e.g., [10]), in software (STM, e.g., [27]), and in combination (HybridTM, e.g., [28]). HTM has the lowest overhead, but transactions are limited in space and time. STM does not have such limitations, but has higher overhead. HybridTM avoids these limitations. Given STM's hardware-independence, which is a compelling advantage, we focus on STM. STM implementations can be classified into three categories: *library-based*, *compiler-based*, and *virtual machine-based*.

- Library-based STMs add transactional support without changing the underlying language. Instrumentation is used to transparently add transactional code to the atomic sections (e.g., begin, transactional reads/writes, commit).
- Compiler-based STMs (e.g., [25], [23]) support implicit transactions transparently by adding new language constructs (e.g., `atomic`). The compiler then generates transactional code that calls the underlying STM library. Compiler-based STMs can optimize the generated code and do overall program optimization.
- VM-based STMs, which have been less studied, include [20], [9], [38], [1]. Transactional support is implemented inside the JVM to get the benefits of a VM-managed environment.

ByteSTM is built by modifying Jikes RVM [3], a Java research virtual machine implemented in Java, using the optimizing compiler. ByteSTM is implicitly transactional: the program only specifies the start and end of a transaction and all memory operations (loads and stores) inside these boundaries are implicitly transactional. This simplifies the code inside the atomic block and also eliminates the need for making a transactional version of each memory load/store instruction, thereby keeping the number of added instructions minimal. When `xBegin` (i.e., the transaction's beginning) is executed, the thread enters transactional mode. In this mode, all writes are isolated and execution of instructions proceeds optimistically until `xCommit` (i.e., the transaction's commit) is executed. At that point, the transaction is checked against other concurrent transactions for conflicts. If there are no conflicts, the transaction is allowed to commit and only at this point do all transaction modifications become externally visible to other transactions. If the commit fails, all modifications are discarded and the transaction restarts from the beginning.

ByteSTM monitors memory accesses at the field level, not at the object level. Field-based granularity scales well and eliminates false conflicts resulting from two transactions changing different fields of the same object [27].

A. Metadata

Working at the VM level allows changing thread headers without modifying program code. For each thread that executes transactions, the metadata added includes the read-set, the write-set, and other STM algorithm-specific metadata. Metadata added to the thread header is used by all transactions executed in the thread. Since each thread executes one transaction at a time, there is no need to create new metadata for each transaction, allowing reuse. Also, accessing a thread's header is faster than using Java's `ThreadLocal` abstraction.

B. Memory Model

At the VM-level, the physical memory addresses of each object's fields can be easily obtained. Since ByteSTM is field-based, the address of each field is used to track memory reads and writes. A conflict occurs only if two transactions modified the same field of an object. Since arrays are objects in Java, memory accesses to arrays are tracked at the element level, which eliminates unnecessary aborts.

An object instance's field's absolute address equals the object's base address plus the field's offset. A static object's field's absolute address equals the global static memory space's address plus the field's offset. Finally, an array element's absolute address equals the array's address plus the element's index in the array (multiplied by the element's size). Thus, our memory model is simplified as: base object plus an offset for all cases.

Using absolute addresses is limited to non-moving GCs only (i.e., a GC that releases unreachable objects without moving reachable objects, like the mark-and-sweep GC). In order to support moving GCs, a field is represented by its base object and the field's offset within that object. When the GC moves an object, only the base object's address is changed. All offsets remain the same. ByteSTM's write-set is part of the GC root-set. Thus, the GC automatically changes the saved base objects' addresses as part of its reference-updating phase.

To simplify how the read-set and write-set are handled, we use a unified memory access scheme. At a memory load, the information needed to track the read includes the base object and the offset within that object of the read field. At a memory store, the base object, the field's offset, the new value, and the size of the value are the information used to track the write. When data is written back to memory, the write-set information (base object, offset, value, and length of the location) is used to store the committed values correctly. This abstraction also simplifies the code, as there is now no need to differentiate between different data types because they are all handled as a sequence of bytes in the memory. The result is simplified code that handles all data types in the same way, yielding faster execution.

ByteSTM has been designed to natively support opacity [17]. In fact, when an inconsistent read is detected in a transaction, the transaction is immediately aborted. Local variables are then restored and the transaction is restarted by throwing an exception. The exception is caught just before the end of the transaction loop so that the loop continues again. Note that throwing an exception is not expensive if the exception object is preallocated, eliminating the overhead of creating a stack trace every time the exception is thrown. A stack trace is not required for this exception object because it is used only for doing a long jump. The result is similar to `setjmp/longjmp` in C.

C. Garbage Collector

One major drawback of implementing STM for Java (or any managed language) is the GC [29]. STM uses metadata to keep track of transactional reads and writes, which requires allocating memory for the metadata and then releasing it when not needed. Frequent memory allocation (and implicit deallocation) forces the GC to run more frequently to release unused memory, increasing STM overhead.

Some STM implementations solve this problem by reducing memory allocations and recycling allocated memory. For example, [27] uses object pooling, wherein objects are allocated from and recycled back to a pool of objects (with the heap used when the pool is exhausted). However, allocation is still done through the Java memory system and the GC checks if the pooled objects are still referenced.

Since ByteSTM is integrated into the VM, its memory allocation and recycling is done outside the control of the Java memory system: memory is directly allocated and recycled. STM's memory requirement, in general, has a specific lifetime. When a transaction starts, it requires a specific amount of metadata, which remains active for the transaction's duration. When the transaction commits, the metadata is recycled. Thus, manual memory management does not increase the complexity or overhead of the implementation.

The GC causes another problem for ByteSTM, however. ByteSTM stores intermediate changes in a write buffer. Thus, the program's newly allocated objects will not be referenced by the program's variables. The GC scans only the program's stack to find objects that are no longer referenced. Hence, it will not find any reference to the newly allocated objects and will recycle their memory. When ByteSTM commits a transaction, it will therefore be writing a dangling pointer. We solve this problem by modifying the behavior of adding an object to the write-set. Instead of storing the object address in the write-set entry value, the object is added to another array (i.e., the "objects array"). The object's index in the objects array is stored in the write-set entry value. Specifically, if an object contains another object (e.g., a field that is a reference), we cannot save the field value as a primitive type (e.g., the absolute address) since the address can be changed by the GC. The field value is therefore saved as an object in the objects array, which is available to the set of roots that the GC scans. The write-set array is another source of roots. So, the write-set contains the base objects and the objects array contains the object fields within the base objects. This prevents the GC from reclaiming the objects. Our approach is compatible with all GCs available in Jikes RVM and we believe that this approach is better than modifying a specific GC.

D. STM Algorithms

ByteSTM's modular architecture allows STM algorithms to be easily "plugged in." We implemented three algorithms: TL2 [12], RingSTM [37], and NOrec [11]. Our rationale for selecting these three is that they are well-known in literature. Additionally, they cover different points in the performance/workload tradeoff space: TL2 is effective for long transactions and a moderate number of reads, and it scales well with a large number of writes; RingSTM is effective for transactions with a high number of reads and a small number of writes; NOrec has better performance with a small number of cores and has no false conflicts since it validates by value.

Plugging a new algorithm into ByteSTM is straightforward: one needs to implement read barriers, write barriers, transaction start, transaction end, and any other algorithm-specific helper methods. All these methods are in one class, "STM.java". No prior knowledge of Jikes RVM is required and porting a new algorithm to ByteSTM requires only understanding the ByteSTM framework.

E. Experimental Evaluation

To understand how ByteSTM, a VM-level STM, stacks up against non VM-level STMs, we conducted an extensive experimental study. The performance study also wanted to investigate whether any performance gain from a VM-level implementation is *algorithm-independent*. Thus, we compared ByteSTM against non-VM STMs, with the same algorithm inside the VM versus "outside" it. Our competitor non-VM STMs include Deuce, ObjectFabric, Multiverse, and JVSTM.

We used a 48-core machine (four AMD Opteron Processors, each with 12 cores running at 1700 MHz) with 16 GB of RAM. The machine ran Ubuntu Linux 10.04 LTS 64-bit. Our test applications included both micro-benchmarks (i.e., data structures) and a macro-benchmark (i.e., STAMP [8]).

1) Micro-Benchmarks: Each data structure was used to implement a sorted integer set interface with set size 256 and set elements in the range 0 to 65536. Writes represented add and remove operations, and they kept the set size approximately constant during the experiments. Different ratios of writes and reads were used to measure performance under different levels of contention: 20% and 80% writes. We also varied the number of threads in exponential steps up to 48.

Linked-list operations were characterized by a high number of reads (ranging from 70 at low contention to 270 at high contention), caused by traversing the list from the head to the required node, and only a few writes (about 2). This resulted in long transactions. Moreover, we observed that such transactions suffer from a high number of aborts (our abort ratio was from 45% to 420%), since each transaction keeps all visited nodes in its read-set and any modification to those nodes by another transaction's add or remove operation will abort the transaction.

Figure 1 shows the results. ByteSTM has three curves: RingSTM, TL2, and NOrec. In all cases, ByteSTM/NOrec achieved the best performance because it uses an efficient read-set data

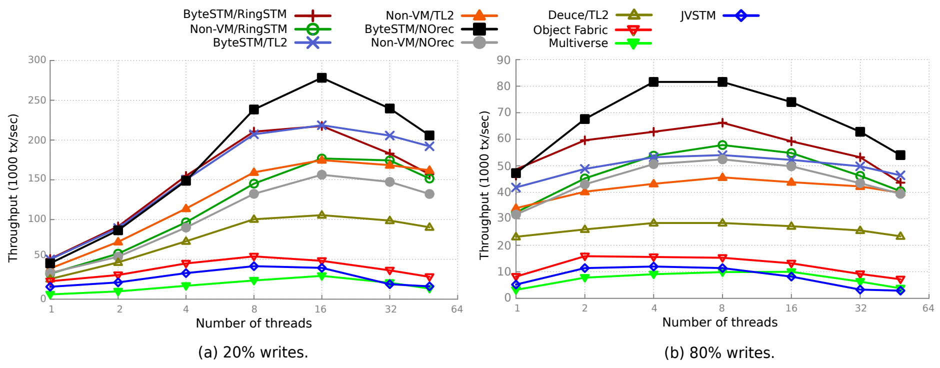

Fig. 1

Throughput for Linked-List. Higher is better.

structure based on open addressing hashing. ByteSTM/RingSTM came next because it has no read-set and uses a bloom filter as a read signature, with the performance being affected by the bloom filter's false positives. This was followed by ByteSTM/TL2, which was affected by its sequential read-set. Deuce's performance was the best between other STM libraries. Other STMs performed similarly, and all of them had very low throughput. Non-VM implementations of each algorithm performed in a similar manner but with lower throughput. ByteSTM outperformed non-VM implementations by 15–70%.

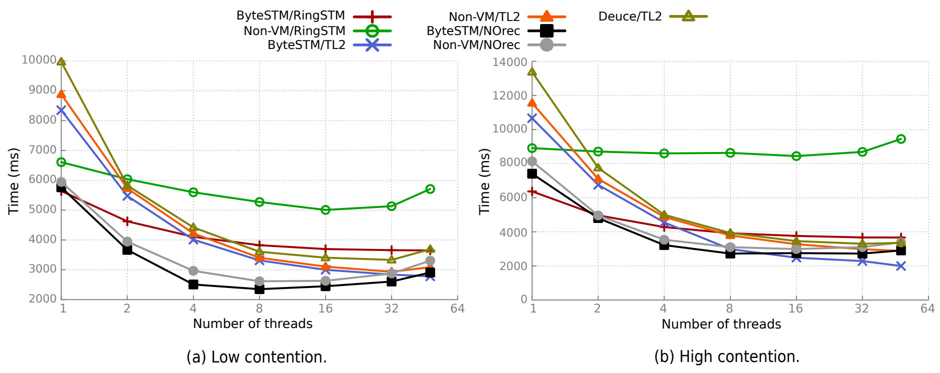

Fig. 2

Execution time under Vacation. Lower is better.

2) Macro-Benchmark: Vacation, an application from the STAMP suite, has medium-length transactions, medium read-sets, medium write-sets, and long transaction times (compared with other STAMP benchmarks). We conducted two experiments: low contention and high contention. ByteSTM/NOrec had the best performance under both low and high contention conditions. The efficient read-set implementation contributed to its performance. However, it did not scale well: ByteSTM/RingSTM suffered from a high number of aborts due to false positives and long transactions, so it started with good performance but degraded quickly. ByteSTM outperformed non-VM implementations by an average of 15.7% in low contention and 18.3% in high contention.

III. Disjoint-access parallelism: impossibility, possibility, and cost of STM implementations

Disjoint-access parallelism (or DAP) [26] is a long-studied property that assesses the ability of a STM implementation to avoid any contention on shared objects (also called *base objects*) between transactions that access disjoint data sets.

The existing STM literature has established that having DAP and serializability of committed transactions in the same STM implementation is impossible under certain conditions for read-only transactions or different progress guarantees [4], [34], [16], [14], [7]. In particular, Attiya et al. [4] proved that a STM cannot be weak DAP (a weaker version of the original DAP [26]) while ensuring minimal progressiveness [19] for write transactions (a progress condition weaker than obstruction-freedom [22]) and providing invisible and wait-free read-only transactions if the correctness guarantee is (Strict) Serializability [6]. More recently, Bushkov et al. [7] proved the impossibility of implementing a strict DAP [16] (a stronger version of the original DAP [26]) STM that guarantees obstruction-freedom and Weak Adaptive Consistency, which is weaker than Serializability.

Unfortunately, the several consistency criteria that have been proposed for STM, such as *opacity* [17], *virtual world consistency* [24], and *TMS1* [13], require at least serializability of committed transactions. However, the constraint for read-only transactions to never change the status of memory, i.e., being *invisible*, as well as their ability to always commit in a finite number of steps, i.e., being *wait-free*, are desirable requirements for enhancing the performance of STM in the case of read-dominated workloads. Therefore, this project pursued the objective of finding the strongest correctness and progress guarantees that a DAP STM algorithm can ensure while having invisible and wait-free read-only transactions. In addition, for those guarantees, the project studied unavoidable costs to consider in terms of both time and space complexity.

In this context, we proved two novel impossibility results that have to be taken into account while designing a DAP STM. Specifically, we proved that if one defines as target correctness criterion *any* criterion that includes Real-Time Order (RTO), i.e., guaranteeing that non-concurrent transactions appear as though they were executed in the order of their commits, then it is impossible to ensure wait-free read-only transactions, obstruction-free write transactions, and the weakest form of DAP. We also proved that, even when assuming weakly progressive write transactions [18], we still have an impossibility result if we enforce invisibility for read-only transactions.

These results highlight the need for relaxing RTO in order to implement any DAP STM that guarantees wait-free and invisible read-only transactions. Indeed, we showed that, by adopting a weaker variant of RTO, i.e., Witnessable Real-Time Order (WRTO), which only enforces RTO among directly conflicting transactions, it is possible to design a STM with wait-free and invisible read-only transactions. In particular, we proposed the first STM implementation that guarantees the strongest variant of DAP, strong progressiveness for write transactions, and wait-free and invisible read-only transactions. This STM provides WRTO and an interesting consistency criterion strictly weaker than opacity, i.e., Extended Update Serializability [2] (EUS). EUS guarantees serializability of committed write transactions, and it ensures that each transaction observes a state produced by a serializable schedule.

Despite the theoretical relevance of this algorithm, the proposed STM comes with overheads, which can hinder its practical relevance. However, we proved that these costs are actually necessary, by deriving two lower bounds on the space and time complexity of any DAP STM that guarantees wait-free and invisible read-only transactions, WRTO, and obstruction-freedom or weak progressiveness for write transactions, when considering a consistency criterion strictly weaker than EUS, i.e., Consistent View [2]. Informally, Consistent View allows a specific type of non-serializable schedule, but it ensures that transactions read from a causally consistent snapshot and prevents observing the effects of aborted transactions.

A. Impossibility Results on Disjoint-Access Parallel STM

The first theoretical result of this project regards the possibility of having a DAP STM that guarantees RTO. This is a fundamental question because it only focuses on the correctness of non-concurrent transactions, which should appear as executed in the order defined by their commits (by preserving RTO indeed), and the result is independent of any other correctness guarantee defined for concurrent transactions. We proved that it is impossible to design such a STM if that has to also guarantee wait-free read-only transactions and obstruction-free write transactions.

The result is independent of the visibility of read-only transactions, therefore also proving that there cannot exist any STM that meets the lower bound defined in [4] if write transactions are obstruction-free. Indeed, if on one side the lower bound defines a necessary condition on the number of write operations a wait-free read-only transaction has to apply in a Strict Serializable and DAP STM, on the other side our result states that those write operations are not sufficient to guarantee RTO and hence Strict Serializability (since it includes RTO).

We also investigated whether or not we can combine all the properties above by considering weakly progressive write transactions (namely, write transactions abort only if they encounter a conflict). In this case we proved that the impossibility of combining RTO and wait-free read-only transactions still holds if read-only transactions are invisible.

The intuition to prove the two impossibility results is the following: we showed that any DAP STM that guarantees WRTO must necessarily accept a history that violates RTO between two non-conflicting transactions.

B. A Strict Disjoint-Access Parallel STM

Since our impossibility results proved that RTO cannot be guaranteed by a DAP STM that provides wait-free and invisible read-only transactions, we also investigated relaxing RTO in order to have such a STM. The result is a STM obtained by adopting Witnessable Real-Time Order (WRTO): it is possible to implement a STM that guarantees the strongest variant of DAP, wait-free and invisible read-only transactions, progressive write transactions, and a correctness criterion whose semantics are very close to those provided by opacity or virtual world consistency. This consistency criterion, known in literature as Extended Update Serializability (EUS) [2], [33], guarantees the serializability of the history of committed write transactions — hence ensuring that the state of the STM is updated without anomalies. Further, analogously to opacity, virtual world consistency, and TMS1, EUS provides Consistent View.

Intuitively, the above properties are achieved as follows. Committed write transactions can be guaranteed to be serializable without sharing any global information and only leveraging metadata associated with transactional objects by adopting a scheme similar to the DAP version of TL2 [5]. On the other hand, Consistent View and WRTO can be ensured by combining a multi-version scheme that allows a read operation by a transaction T to never incur wait conditions and to always return the right version v such that v was not committed by a transaction T' that causally follows T, i.e., v is a correct state observable by T. This is achievable without sharing any global information (e.g., a global clock, which would violate DAP) and without applying any modification during the execution of a read operation (invisible read-only transactions) by just exploiting metadata associated with the committed versions in order to detect the causal dependencies of commit events.

C. Time Complexity of Disjoint-Access Parallel STM implementations

Despite the theoretical relevance of the proposed STM, we also proved that it comes with a non-negligible overhead: a read operation of a read-only transaction may accomplish $k \cdot N_o$ steps in order to return the correct value, where k is the number of versions of a transactional object and N_o is the total number of transactional objects. We proved that this cost is necessary even considering only Consistent View and WRTO. In particular, we investigated the costs a DAP STM has to pay if it guarantees wait-free and invisible read-only transactions when the correctness guarantee is Consistent View. Roughly, this means that every read operation does not observe any incorrect state as long as write transactions produce correct states. We also suppose that the STM guarantees WRTO in order to rule out any trivial implementation of wait-free read-only transactions in which read operations only observe the initial versions of the transactional objects.

The intuition behind the result is that such a STM cannot always ensure a constant cost for handling any read operation because a read cannot rely on any shared timestamp to determine the correct state to be observed (as for TL2 [12] or LSA [35]). In fact, if that were the case, then the STM would trivially violate DAP. In particular, what we show is that the maximum number of steps performed by any read operation of read-only transactions for determining whether a version can be observed without violating Consistent View is $\Omega(N_o)$.

D. Space Complexity of Disjoint-Access Parallel STM implementations

In addition to showing the inherent temporal costs a DAP STM must pay in order to execute invisible and wait-free read-only transactions, and if it has to also combine at least Consistent View and WRTO, we also focused on the memory occupancy such a STM would require. That is because the STM proposed in this project requires each version of a transactional object to keep a vector clock as big as the maximum number of concurrent processes (N_p). The outcome of our investigation was analogous to the one obtained for the time complexity: in the common case where N_p is lesser than the number of objects (N_o), that spatial cost is necessary.

Furthermore, in general we proved a lower bound that holds assuming Consistent View as correctness guarantee, WRTO, and either weak progressiveness or obstruction-freedom for write transactions. If a DAP STM has to jointly ensure those properties then the space complexity for each version of a transactional object is $\Omega(m)$, where $m = min(N_o, N_p)$.

To prove that, we used an innovative technique that shows an equivalence between the problem of detecting cycles in the conflict graph of a history generated by DAP STM and determining causality in a distributed message passing system. The intuition behind the proof is that whenever a read-only transaction executes a read operation, it needs to detect whether that operation creates a cycle with one anti-dependence edge (which would violate Consistent View) in the conflict graph associated with the current history. Due to the existence of the DAP requirement, this check has to be performed without indiscriminately accessing all the information associated with the conflict graph, but only extracting this information via the base objects associated with the accessed transactional objects.

IV. Summary of Project's Deliverables: Software and Publications

The results accomplished by this project have been publicly released as open-source software and research papers published at international conferences. In the following we summarize them:

- Software:
  – ByteSTM is open-sourced and is freely available at `http://hydravm.org/bytestm`.
- Publications:
  – M. Mohamedin, B. Ravindran, and R. Palmieri. *Bytestm: Virtual machine-level java software transactional memory*. In Coordination Models and Languages, 15th International Conference, COORDINATION, Florence, Italy, June 3-5, 2013.
  – M. Mohamedin. ByteSTM: Java Software Transactional Memory at the Virtual Machine Level. Master's thesis, Virginia Tech, 2012. Available at `http://scholar.lib.vt.edu/theses/available/etd-02222012-091827/`.
  – S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In ACM Symposium on Principles of Distributed Computing, PODC, Donostia-San Sebastian, Spain, July 21-23, 2015.

References

[1] A. Adl-Tabatabai et al. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 2003.

[2] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999.

[3] B. Alpern, S. Augart, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.

[4] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory Comput. Syst.*, 49(4):698–719, 2011.

[5] H. Avni and N. Shavit. Maintaining Consistent Transactional States Without a Global Clock. In *SIROCCO*, pages 131–140, 2008.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. The PCL Theorem. Transactions cannot be Parallel, Consistent and Live. In *SPAA*, pages 178–187, 2014.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.

[9] B. Carlstrom, A. McDonald, et al. The Atomos transactional programming language. *ACM SIGPLAN Notices*, 41(6):1–13, 2006.

[10] D. Christie, J. Chung, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys*, pages 27–40, 2010.

[11] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78. ACM, 2010.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, pages 194–208, 2006.

[13] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.

[14] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *PODC*, pages 115–124, 2012.

[15] V. Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 1–10, 2015.

[16] R. Guerraoui and M. Kapalka. On Obstruction-free Transactions. In *SPAA*, pages 304–313, 2008.

[17] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[18] R. Guerraoui and M. Kapalka. The Semantics of Progress in Lock-based Transactional Memory. In *POPL*, 2009.

[19] R. Guerraoui and P. Romano. *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Springer, 2015.

[20] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[21] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition.* Morgan and Claypool Publishers, 2nd edition, 2010.

[22] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues As an Example. In *ICDCS*, pages 522–529, 2003.

[23] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Workshop on Memory system performance and correctness*, pages 82–91, 2006.

[24] D. Imbs and M. Raynal. Virtual World Consistency: A Condition for STM Systems (with a Versatile Protocol with Invisible Read Operations). *Theoretical Computer Science*, 444:113–127, July 2012.

[25] Intel Corporation. Intel C++ STM Compiler, 2009. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`.

[26] A. Israeli and L. Rappoport. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *PODC*, pages 151–160, 1994.

[27] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[28] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.

[29] F. Meawad et al. Collecting transactional garbage. In *TRANSACT*, 2011.

[30] M. Mohamedin. ByteSTM: Java Software Transactional Memory at the Virtual Machine Level. Master's thesis, Virginia Tech, 2012. Available at `http://scholar.lib.vt.edu/theses/available/etd-02222012-091827/`.

[31] M. Mohamedin, B. Ravindran, and R. Palmieri. Bytestm: Virtual machine-level java software transactional memory. In *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Florence, Italy, June 3-5, 2013. Proceedings*, pages 166–180, 2013.

[32] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *ACM Symposium on Principles of Distributed Computing, PODC '15, Donostia-San Sebastian, Spain, July 21-23, 2015*, 2015.

[33] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *ICDCS*, pages 455–465, 2012.

[34] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, 2010.

[35] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *DISC*, pages 284–298, 2006.

[36] M. M. Saad, M. Mohamedin, and B. Ravindran. HydraVM: extracting parallelism from legacy sequential code using STM. In *HotPar*. USENIX, 2012. `hydravm.org`.

[37] M. F. Spear, M. M. Michael, and C. von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[38] A. Welc et al. Transactional monitors for concurrent objects. In *ECOOP*, 2004.

1.

1. Report Type

Final Report

Primary Contact E-mail
Contact email if there is a problem with the report.

binoy@vt.edu

Primary Contact Phone Number
Contact phone number if there is a problem with the report

5402313777

Organization / Institution name

Virginia Tech

Grant/Contract Title
The full title of the funded effort.

Virtual Machine-level Software Transactional Memory: Principles, Techniques, and Implementation

Grant/Contract Number
AFOSR assigned control number. It must begin with "FA9550" or "F49620" or "FA2386".

FA9550-14-1-0143

Principal Investigator Name
The full name of the principal investigator on the grant or contract.

Binoy Ravindran

Program Manager
The AFOSR Program Manager currently assigned to the award

Dr. Kathleen Kaplan

Reporting Period Start Date

07/01/2014

Reporting Period End Date

06/30/2015

Abstract

Software transactional memory (STM) has emerged as an easy to program synchronization abstraction for multicore computer architectures. But performance of current STM frameworks are inferior and heavily influenced by infrastructure resource management (e.g., operating system scheduling, garbage collection, memory allocation).

The project’s first major result is ByteSTM (COORDINATION’13, TRANSACT’13), a virtual machine-level Java STM implementation that is built by extending the Jikes RVM. The project modified Jikes RVM’s optimizing compiler to transparently support implicit transactions. Being implemented at the VM-level, which enables direct memory accesses, ByteSTM avoids Java garbage collection overhead by manually managing memory for transactional metadata, and provides pluggable support for implementing different STM algorithms. Three well-known STM algorithms have been integrated into ByteSTM: TL2, NOrec, and RingSTM. The programmer can switch among them by simply changing one configuration parameter. This enables easy evaluation of application performance under different STM concurrency controls. The project’s experimental studies revealed throughput improvement over other non-VM STMs by 6–70% on micro-benchmarks and by 7–60% on macro-benchmarks. ByteSTM is open-source, publicly available (http://hydravm.org/bytestm/), and is used by the TM community.

The project's second major result is a set of possibility and impossibility results on the feasibility of ensuring the Disjoint-Access Parallelism (DAP) property for STM implementations (PODC'15). DAP is one of the most desirable properties for maximizing TM's scalability. The project investigated the possibility and inherent cost of implementing a DAP TM that ensures the two most important properties for maximizing performance of read-dominated workloads, namely, having invisible and wait-free read-only transactions. The project proved that relaxing the Real-Time Order (RTO) is necessary to implement such a TM. This result motivated the project to introduce Witnessable Real-Time Order (WRTO), a weaker variant of RTO that demands enforcing RTO only between directly conflicting transactions. The project demonstrated that adopting WRTO makes it possible to design a strictly DAP TM with invisible and wait-free read-only transactions, while preserving strong progressiveness for write transactions and an isolation level known in the literature as Extended Update Serializability. Finally, the project shed light on the inherent inefficiency of DAP TM implementations that have invisible and wait-free read-only transactions, by establishing lower bounds on the time and space complexity of such TMs.

Distribution Statement

This is block 12 on the SF298 form.

Distribution A - Approved for Public Release

Explanation for Distribution Statement

If this is not approved for public release, please provide a short explanation. E.g., contains proprietary information.

SF298 Form

Please attach your SF298 form. A blank SF298 can be found here. Please do not password protect or secure the PDF The maximum file size for an SF298 is 50MB.

AFD-070820-035.pdf

Upload the Report Document. File must be a PDF. Please do not password protect or secure the PDF . The maximum file size for the Report Document is 50MB.

report-v2.pdf

Upload a Report Document, if any. The maximum file size for the Report Document is 50MB.

Archival Publications (published) during reporting period:

M. Mohamedin, B. Ravindran, and R. Palmieri. "Bytestm: Virtual machine-level Java software transactional memory". In Coordination Models and Languages, 15th International Conference, COORDINATION, Florence, Italy, June 3-5, 2013.

M. Mohamedin. "ByteSTM: Java Software Transactional Memory at the Virtual Machine Level". Master's thesis, ECE Department, Virginia Tech, 2012. Available at http://scholar.lib.vt.edu/theses/available/etd-02222012-091827/.

S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. "Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations". In ACM Symposium on Principles of Distributed Computing, PODC, Donostia-San Sebastian, Spain, July 21-23, 2015.

Changes in research objectives (if any):

Change in AFOSR Program Manager, if any:

Extensions granted or milestones slipped, if any:

AFOSR LRIR Number

LRIR Title

Reporting Period

Laboratory Task Manager

Program Officer

Research Objectives

Technical Summary

Funding Summary by Cost Category (by FY, $K)

	Starting FY	FY+1	FY+2
Salary			
Equipment/Facilities			
Supplies			
Total			

Report Document

Report Document - Text Analysis

Report Document - Text Analysis

Appendix Documents

2. Thank You

E-mail user

Aug 10, 2015 18:30:39 Success: Email Sent to: binoy@vt.edu